Using Model Checking Techniques for Symbolic Synthesis of Distributed Programs *

Fuad Abujarad Borzoo Bonakdarpour Sandeep S. Kulkarni
Department of Computer Science and Engineering
Michigan State University
East Lansing, MI 48824, USA
Email: {abujarad,borzoo,sandeep}@cse.msu.edu
Web: http://www.cse.msu.edu/~{abujarad,borzoo,sandeep}

Abstract

Given the non-determinism and race conditions in distributed programs, the ability to provide assurance about them is crucial. Our work focuses on incremental synthesis where we modify existing (fault-intolerant) distributed programs to add fault-tolerance. We concentrate on reducing the complexity of such synthesis using techniques –symmetry and parallelism— from model checking. We apply these techniques in the context of deadlock resolution. In particular, incremental synthesis requires removal of certain program actions that could violate safety in the presence of faults and such removal may eliminate all outgoing transitions from some states rendering them to be deadlock states. We focus on reducing the complexity of resolving such deadlock states using symmetry and/or parallelism. We show that these approaches provide a significant speedup separately as well as together.

Keywords: Program transformation, Program synthesis, Multi-core algorithm, Symmetry, Distributed programs.

1 Introduction

In this paper, we target the issues in the automated design of a fault-tolerant distributed program from its fault-intolerant version. Such automated revision is highly desirable since it enables system designers to automatically and incrementally add properties to distributed programs. One of the problems in the tools for providing automated revision is that the time and space complexity of the revision algorithms is high. In our previous work, we have shown that symbolic synthesis techniques [4] are very useful in reducing the complexity. In particular, we proposed a set of symbolic (BDD[1]-based) techniques for adding fault-tolerance to existing moderate-sized, fault-intolerant distributed programs. We showed that symbolic techniques improved the performance by several orders of magnitude. Moreover, experimental results exhibited feasibility of synthesis of programs with state space of size 10^{30} and beyond. One way to reduce the complexity further is to integrate advances from model checking, as incremental synthesis involves several tasks that are also considered in model checking. We consider two approaches from model checking: (1) the use of symmetry and (2) the parallelism of the algorithm with multiple processors/cores.

To understand the use of symmetry, we observe that multiple processes in a distributed program are symmetric in nature, i.e., their actions are similar (except for the renaming of variables).

*This work was partially sponsored by NSF CAREER CCR-0092724 and ONR Grant N00014-01-1-0744.

[1]Ordered Binary Decision Diagrams [6] represent Boolean formulae as directed acyclic graphs making testing of functional properties such as satisfiability and equivalence straightforward and extremely efficient.

Report Documentation Page

Form Approved
OMB No. 0704-0188

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED
2008		**00-00-2008 to 00-00-2008**

4. TITLE AND SUBTITLE	5a. CONTRACT NUMBER
Using Model Checking Techniques for Symbolic Synthesis of Distributed Programs	5b. GRANT NUMBER
	5c. PROGRAM ELEMENT NUMBER
6. AUTHOR(S)	5d. PROJECT NUMBER
	5e. TASK NUMBER
	5f. WORK UNIT NUMBER
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) **Michigan State University,Department of Computer Science and Engineering,East Lansing,MI,48824**	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)
	11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT

Given the non-determinism and race conditions in distributed programs, the ability to provide assurance about them is crucial. Our work focuses on incremental synthesis where we modify existing (fault-intolerant) distributed programs to add fault-tolerance. We concentrate on reducing the complexity of such synthesis using techniques ?symmetry and parallelism? from model checking. We apply these techniques in the context of deadlock resolution. In particular, incremental synthesis requires removal of certain program actions that could violate safety in the presence of faults and such removal may eliminate all outgoing transitions from some states rendering them to be deadlock states. We focus on reducing the complexity of resolving such deadlock states using symmetry and/or parallelism. We show that these approaches provide a significant speedup separately as well as together.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			
unclassified	**unclassified**	**unclassified**	**Same as Report (SAR)**	**12**	

Thus, if we find recovery transitions for a process then we can utilize symmetry to identify other recovery transitions that should also be included for other processes in the system. Likewise, if some transitions of a process violate safety in the presence of faults then we can identify similar transitions of other processes that would also violate safety. If the cost of identifying these similar transitions with the knowledge of symmetry among processes is less than the cost of identifying these transitions explicitly then the use of symmetry will reduce the overall time required for synthesis.

To understand the use of parallelism, we note that several tasks in the synthesis could be expedited if we utilize a parallel computation where multiple threads cooperate to compute recovery transitions and/or transitions to be removed for safety violation. One of the problems in automated revision is the *group* computation that is caused due to inability of a process in a distributed program to read all variables. In particular, consider the case where a recovery transition of the form 'if (condition) then statement' is added, it corresponds to several program transitions since 'condition' and 'statement' cannot include variables that a process cannot read. We use parallelism to speedup the group computation. In particular, we split the computation of the groups among the available threads/cores. Each thread performs partial computation and the results are collected to obtain the required group of transitions.

Contribution of the paper. We present an algorithm for utilizing symmetry and exploiting parallelism. We show that this algorithm significantly improves performance over previous implementations. For example, in the case of *Byzantine agreement* (*BA*) [16] with 25 processes time for synthesis with sequential algorithm was $1,632s$. With symmetry alone, synthesis time was reduced to $188s$ (8.7 times better). With parallelism (8 threads), synthesis time was reduced to $467s$ (3.5 times better). When we combined both symmetry and parallelism together, the total synthesis time was reduced to $107s$ (more than 15.2 times better).

Organization of the paper. The rest of this paper is organized as follows: In Section 2, we define the problem statement for fault-tolerance addition. In Section 3, we describe the Byzantine agreement problem that we use in describing the use of symmetry and parallelism. In Section 4, we present our approach for expediting the synthesis of fault-tolerant programs with the use of symmetry and parallelism. Section 5 presents our experimental results in two case studies. In Section 6, we present the related work. Finally, the conclusion and future work are described in Section 7.

2 Distributed Programs and Specifications

In this section, we define the problem statement for adding fault-tolerance. We begin with a fault-intolerant program, say p, that is correct in the absence of faults. We let p be specified in terms of its state space, S_p, and a set of transitions, $\delta_p \subseteq S_p \times S_p$. Whenever it is clear from the context, we use p and its transitions δ_p interchangeably. A sequence of states, $\langle s_0, s_1, ...\rangle$ is a computation of p iff (1) $(\forall j : j > 0 : (s_{j-1}, s_j) \in p)$, i.e., in each *step* of this sequence, a transition of p is executed, and (2) if the sequence is finite and terminates in s_j then $\forall s' :: (s_j, s') \notin p$, i.e., a computation is finite only if it reaches a state from where the program does not have any outgoing transition. A special subset of S_p, say S, identifies an *invariant* of p. By this we mean that if a computation of p begins in a state where S is true, then (1) S is true at all states in that computation and (2) the computation is *correct*. Note that the notion of this *correctness* has to deal with the *fault-intolerant* program that is assumed to be correct.

The goal of an algorithm that adds fault-tolerance is to begin with a program p and its invariant S to derive the fault-tolerant program, say p', and its invariant, say S'. Clearly, one additional input to such an algorithm is f, the class of faults to which tolerance is to be added. Faults are also specified as a subset of $S_p \times S_p$. Note that this allows modeling of different types of faults, such as transients, Byzantine (see Section 3), crash faults, etc. Yet another input to the algorithm for adding fault-tolerance is a safety specification, say $spec_{bt}$, that should not be violated in the

presence of faults. We let $spec_{bt}$ also be specified by a set of transitions, i.e., $spec_{bt}$ is a subset of $S_p \times S_p$[2]. Thus, it is required that in the presence of faults, the program should not execute a computation from $spec_{bt}$.

Now we define the problem of adding fault-tolerance where the input is program p, invariant S, faults f, and safety specification $spec_{bt}$. Since our goal is to add fault-tolerance only, we require that no new computations are added in the *absence* of faults. Thus, if the output after adding fault-tolerance is program p' and invariant S', then S' should not include any states that are not in S; without this restriction, p' can begin in a state from where the correctness of p is unknown. Likewise, if (s_0, s_1) is a transition of p' and $s_0 \in S'$ then (s_0, s_1) must also be a transition of p; without this restriction, p' will have new computations in the absence of faults. Also, if p' has no outgoing transition from state $s_0 \in S'$, then it must be the case that p also has no outgoing transitions from s_0; without this restriction, p' may stop in a state that had no correspondence with p.

Additionally, p' should be fault-tolerant. Thus, during the computation of p', if faults from f occur then the program may be perturbed to a state outside S'. Just like the invariant captured the boundary up to which the program can reach in the *absence* of faults, we can identify a boundary upto which the program can reach in the *presence* of faults. Let this boundary (denoted by fault-span) be T. Thus, if any transition of p or f begins in a state where T is true, then it must terminate in a state where T is true. Moreover, if p' is permitted to execute for a long enough time without perturbation of a fault, then p' should reach a state where its invariant S' is true. Based on this discussion, we define the problem of adding fault-tolerance as follows:

Problem statement 2.1 Given p, S, f and $spec_{bt}$, identify p' and S' such that:

- $(C1)$: Constraints on the invariant
  - $S' \neq \phi$,
  - $S' \Rightarrow S$,
- $(C2)$: Constraints on transitions within invariant
  - $(s_0, s_1) \in p' \ \wedge \ s_0 \in S' \quad \Rightarrow \quad ((s_1 \in S') \wedge (s_0, s_1) \in p)$,
  - $s_0 \in S' \wedge (\forall s_1 :: (s_0, s_1) \notin p') \quad \Rightarrow \quad (\forall s_1 :: (s_0, s_1) \notin p)$, and
- $(C3)$ There exists T such that
  - $S' \Rightarrow T$,
  - $s_0 \in T \wedge (s_0, s_1) \in (p' \cup f) \quad \Rightarrow \quad s_1 \in T \wedge (s_0, s_1) \notin spec_{bt}$
  - $s_0 \in T \wedge \langle s_0, s_1, ... \rangle$ is a computation of $p' \quad \Rightarrow \quad (\exists j : j \geq 0 : s_j \in S')$

3 The Byzantine Agreement Problem

We now illustrate the synthesis problem identified in Section 2 and issues involved in solving it in the context of using the *Byzantine agreement* (denoted BA) problem [16]. In particular, we identify the four inputs used in the problem statement above. These inputs would be used in Section 4 to describe the use of symmetry and parallelism. BA consists of a *general*, say g, and three (or more) *non-general* processes, say j, k, and l. The agreement problem requires that a process copy the decision chosen by the general (0 or 1) and finalize (output) the decision (subject to some constraints). Thus, each process of BA maintains a decision d; for the general, the decision can be either 0 or 1, and for the non-general processes, the decision can be 0, 1, or $\perp$, where the value $\perp$ denotes that the corresponding process has not yet received the decision from the general. Each non-general process also maintains a Boolean variable f that denotes whether that process has finalized its decision. For each process, a Boolean variable b shows whether or not the process is Byzantine; the read/write restrictions, specified later, ensure that a process cannot determine if other processes are Byzantine. Thus, a state of the program is obtained by assigning each variable, listed below, a value from its domain. And, the state space of the program is the set of all possible states.

[2] As shown in [15], permitting more general specifications can significantly increase the complexity and this representation suffices for most practical programs, we model safety specification using a set of transitions

$V = \{d.g\} \cup$ (the general decision variables):{0,1}
$\{d.j, d.k, d.l\} \cup$ (the processes decision variables):{ 0, 1, $\perp$}
$\{f.j, f.k, f.l\} \cup$ (finalized?):{$false$,$true$}
$\{b.g, b.j, b.k, b.l\}$. (Byzantine?):{$false$,$true$}

To concisely describe the transitions of the (fault-intolerant) version of BA, we use guarded commands of the form $g \longrightarrow st$, where g is a predicate involving the above program variables and st updates the above program variables. The command $g \longrightarrow st$ corresponds to the set of transitions $\{(s_0, s_1) : g$ is true in s_0 and s_1 is obtained by *executing* st in state $s_0\}$. Thus, the transitions of a non-general process, say j, is specified by the following two actions:

$$\begin{array}{lllll} BA_{intol_j} :: & BA1_j :: (d.j = \perp) \wedge (f.j = false) \wedge (b.j = false) & \longrightarrow & d.j := d.g \\ & BA2_j :: (d.j \neq \perp) \wedge (f.j = false) \wedge (b.j = false) & \longrightarrow & f.j := true \end{array}$$

Note that the general does not need explicit actions; the action by which the general sends the decision to j is modeled by $BA1_j$. The variables that a non-general process, say j, is allowed to read and write are $R_j = \{b.j, d.j, f.j, d.k, d.l, d.g\}$ and $W_j = \{d.j, f.j\}$, respectively. Observe that this modeling prevents j from knowing whether other processes are Byzantine.

The safety specification of the BA requires *validity* and *agreement*. *Validity* requires that if the general is non-Byzantine, then the final decision of a non-Byzantine, non-general must be the same as that of the general. Additionally, *agreement* requires that the final decision of any two non-Byzantine, non-generals must be equal. Finally, once a non-Byzantine process finalizes (outputs) its decision, it cannot change it. Thus, the following transition predicate forms the safety specification, where p and q range over non-general processes, unprimed variables denote the values in the source states, and primed variables denote the value of the variable in the target state state:

$$\begin{array}{l} SPEC_{bt_{BA}} = (\exists p :: \neg b'.g \wedge \neg b'.p \wedge (d'.p \neq \perp) \wedge f'.p \wedge (d'.p \neq d'.g)) \ \vee \\ \qquad (\exists p, q :: \neg b'.p \wedge \neg b'.q \wedge f'.p \wedge f'.q \wedge (d'.p \neq \perp) \wedge (d'.q \neq \perp) \wedge (d'.p \neq d'.q)) \ \vee \\ \qquad (\exists p :: \neg b.p \wedge \neg b'.p \wedge f.p \wedge ((d.p \neq d'.p) \vee (f.p \neq f'.p))) \end{array}$$

The invariant predicate of the Byzantine agreement program is S_{BA}, where

$$\begin{array}{ll} S_{BA} = & \neg b.g \wedge (\neg b.j \vee \neg b.k) \wedge (\neg b.k \vee \neg b.l) \wedge (\neg b.l \vee \neg b.j) \ \wedge \\ & (\forall p :: \neg b.p \Rightarrow (d.p = \perp \vee d.p = d.g)) \ \wedge \ (\forall p :: (\neg b.p \wedge f.p) \Rightarrow (d.p \neq \perp)) \ \vee \\ & b.g \wedge \neg b.j \wedge \neg b.k \wedge \neg b.l \wedge (d.j = d.k = d.l \ \wedge d.j \neq \perp) \end{array}$$

Thus, the invariant specifies the following constraints: At most one process can be Byzantine. If the general and the non-general j are non-Byzantine then the invariant specifies that $d.j$ can be either $\perp$ or $d.g$. A process cannot finalize its decision if its decision is equal to $\perp$. Finally, for the case where the general is Byzantine, the above invariant states that the fault-intolerant program is correct from states where the decision of all non-generals is equal and it is not equal to $\perp$.

A fault transition can cause a process to become Byzantine, if no other process is initially Byzantine. Also, a fault can change the d and f values of a Byzantine process. The fault transitions that affect a process, say j, of BA are as follows: (We include similar actions for k, l, and g)

$$\begin{array}{lll} F1 :: \neg b.g \wedge \neg b.j \wedge \neg b.k \wedge \neg b.l & \longrightarrow & b.j := true \\ F2 :: b.j & \longrightarrow & d.j, f.j := 0|1, false|true \end{array}$$

where $d.j := 0|1$ means that $d.j$ could be assigned either 0 or 1. In case of the general process, the second action does not change the value of any f-variable.

4 Using Symmetry and Parallelism in Synthesis

In this section, we present our approach for expediting the synthesis of fault-tolerant program with the use of symmetry and parallelism using the input from Section 3. We utilize this approach in the task of resolving deadlock states that are encountered during the synthesis process. Hence, using the example *BA* from Section 3, we first describe deadlock scenarios that occur during synthesis and how symmetry and/or parallelism can help in resolving it. Then we discuss our algorithms for resolving deadlock states. The first algorithm uses symmetry alone. The second uses parallelism alone. Then, we combine both symmetry and parallelism in the third algorithm.

Note that during synthesis, we analyze the effect of faults on the given fault-intolerant program and identify a fault-tolerant program that meets the constraints of Problem 2.1. We illustrate issues involved in the such synthesis next.

- Consider the state where all processes are non-Byzantine, $d.g$ is 0 and the decision of all non-generals is $\perp$. From this state, the program (BA_{intol}) and faults ($F1$ & $F2$) can reach a state, say s_1, where $d.g = d.j = d.k = 0, b.g = true, d.l = 1, f.l = 0$. From such a state, transitions of the fault-intolerant program violate safety if they allow j (or k) and l to finalize their decision. If we remove these safety violating transitions, there are no other transitions from state s_1. In other words, during synthesis, we encounter that s_1 is a deadlock state. One can resolve this deadlock state by simply adding a *recovery* transition that changes the $d.l$ to 0.

- Again, consider the state where all processes are non-Byzantine, $d.g$ is 0 and the decisions of all non-generals are $\perp$. From this state, the program (BA_{intol}) and faults ($F1$ & $F2$) can reach a state, say s_2, where $d.g = d.j = d.k = 0, b.g = true, d.l = 1, f.l = 1$; state s_2 differs from s_1 in the previous case in terms of the value of $f.l$. Unlike s_1 in the previous scenario, since l has finalized its decision, we cannot resolve s_2 by adding safe recovery. One way to resolve this deadlock situation is by eliminating s_2 (i.e., making it unreachable by the program actions). However, since we require that during the elimination of a deadlock state, no new deadlock states be created, the deadlock *elimination* algorithm has to deal with many backtracking steps. In particular, in order to resolve s_2, the algorithm needs to explore the reachability path and remove the transition that allows a process to finalize its decision while there exist two undecided processes.

To maximize the success of the synthesis algorithm, our approach to handle deadlock states is as follows: Whenever possible, we add recovery transition(s) from the deadlock states to a legitimate state. However, if no recovery transition(s) can be added from the deadlock states, we try to eliminate the deadlock states by preventing the program from reaching the deadlock states. In other words, we try to eliminate deadlock states only if adding recovery from them fails.

In this paper, we utilize symmetry and parallelism to expedite these two aspects of deadlock resolution: adding recovery and eliminating deadlock states. We address this problem by three different approaches.

4.1 Symmetry

To describe the use of symmetry, consider the first scenario described above. In this scenario, we resolved the state s_1 by adding a recovery transition. Due to the symmetry of the non-generals, one can observe that we can also add other recovery transitions. For example, if we consider the state $d.g = d.j = d.l = 0, b.g = true, d.k = 1, f.k = 0$, we can add the recovery transition by which $d.k$ changes to 0.

With this observation, if we identify recovery action(s) to be added for one process, we can add the similar actions that correspond to other processes. Therefore, to add recovery, our algorithm does the following: whenever, we find recovery transition(s), we identify other recovery transitions

Pseudocode 1 Add Symmetrical Recovery -

Input: deadlock states ds, invariant S, and unacceptable transitions (including $spec_{bt}$) mt
Output: recovery transitions predicate rec

1: $rec := ds \wedge \langle lyr \rangle'$;
   // $\langle lyr \rangle'$ the set of states to which recovery can be added to ensure recovery to invariant

2: $rec :=$ *Group*(rec, read/write restrictions on i);
   // Select program transition or process i while ensuring read/write restrictions

3: $rec := rec \wedge \neg$*Group*($rec \wedge mt$);
   // Remove transition that violate safety while ensuring distribution restrictions

   // Find similar transitions for other processes
4: **for** $i := 1$ **to** $numberOfProcesses$ **do**
5: $rec := rec \vee$SwapVariables(rec, i);
   // Generate BDDs for other processes by swapping variables based on symmetry
6: **end for**
7: **return** rec;

Pseudocode 2 Group Symmetry -

Input: a set of transitions $trans$.
Output: a group of transitions grp.

1: $grp :=$ *FindGroup*($trans$, read/write restrictions on i);
   // Find the group related to process i transitions while ensuring the read/write restrictions

   // Find similar transitions for other processes
2: **for** $i := 1$ **to** $numberOfProcesses$ **do**
3: $grp := grp \vee$SwapVariables(grp, i);
   // Generate BDDs for other processes by swapping variables based on symmetry
4: **end for**

5: **return** grp;

Figure 1: Pseudocode for Using Symmetry in Resolving Deadlock States

based on symmetry. Then, we add all these recovery transitions to the program being synthesized (cf. Pseudocode 1).

We also apply symmetry for deadlock states elimination. To eliminate a set of deadlock states, we find the set of transitions which if removed from one process, will prevent that process from reaching deadlock states. Then, we use this set of transitions to remove similar transitions from other processes. Therefore, to eliminate deadlock states by removing program transitions, our algorithm does the following: whenever we find a set of transition(s) if removed from one process, will prevent it from reaching a deadlock state. We use symmetry to identify similar transitions for other processes, and we remove these transitions from program transitions (cf. Pseudocode 2).

4.2 Parallelism

In the previous section, we have seen that to resolve deadlock states, we either need (1) to add recovery transitions or (2) to remove transitions to eliminate deadlock states. In both operations, we cannot add/remove the selected transition alone, but we will also need to add/remove the transition *Group*. For example, in the above scenario, when we add a recovery transition from s_1, it corresponds to several recovery transitions where the values of variables that l can read $(d.j, d.k, d.l, d.g, b.l, f.l)$, are fixed but the values of other variables $(b.g, b.j, b.k, f.j, f.k)$ change. Thus, while adding recovery from s_1, we add a group of transitions based on different possible values of $(b.g, b.j, b.k, f.j, f.k)$.

To compute the *Group* associated with a set of transitions, the sequential algorithm will go through many computations for each process, one after another. However, in the parallel algorithm, we split the *Group* computation over the available number of threads. In particular, rather than having one thread find the *Group* for all the processes, we let each thread compute the *Group* for a subset of the processes. The tasks assigned to each thread are fine-grained. Hence, there will be considerable amount of time wasted in the overhead associated with the threads creation/destruction every time the *Group* is computed. Therefore, we let the main thread create the

worker threads at the initialization stage of the synthesis algorithm. The worker threads will stay idle until they are needed. When the main thread needs to find the *Group* of a set of transitions, it will activate the worker threads, through mutexes, to start computing the *Group*. When all worker threads are done the main thread, which was on hold, will be activated to collect the results of all worker threads in one *Group*.

4.3 Symmetry and Parallelism

We use the symmetry as described in section 4.1, except when we compute the similar transitions for other processes, we split the operation over the available threads. Hence, after we find the initial set of transitions (to be added/removed), we split the operation of finding similar transition for other processes among available threads. Once all threads are done, we join their results in one set.

5 Experimental Results

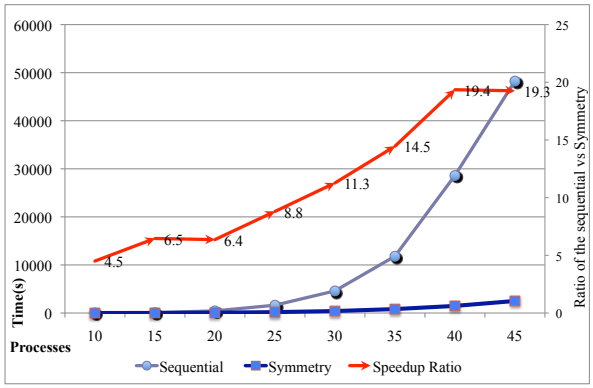
In Subsections 4.1-4.3, we described different approaches to resolve deadlock states in the synthesis of fault-tolerant programs. In Subsections 5.1-5.3, we describe and analyze the respective experimental results. In particular, we describe the results in the context of two classical examples in the literature of distributed computing, namely, the Byzantine agreement (described in Section 3) and the token ring [2]. In both case studies, we find that symmetry and parallelism improve the execution time substantially.

Throughout this section, all experiments are run on a Sun Fire V40z with 4 dual-core Opteron processors and 16 GB RAM. The OBDD representation of the Boolean formulae has been done using the C++ interface to the CUDD package developed at University of Colorado [18]. Throughout this section, we refer to the original implementation of the synthesis algorithm (without symmetry or parallelism) as *sequential* implementation. We refer to the symmetric approach described in Subsection 4.1 as *symmetry* and *X threads* to refer to the parallel algorithm that utilizes X threads. We would like to note that the synthesis time duration differences between the sequential implementation in this paper and the one in [4] is due to other unrelated improvements on the sequential implementation itself. However, the sequential, symmetric, and parallel implementations differ only in terms of the modification described in Section 4.

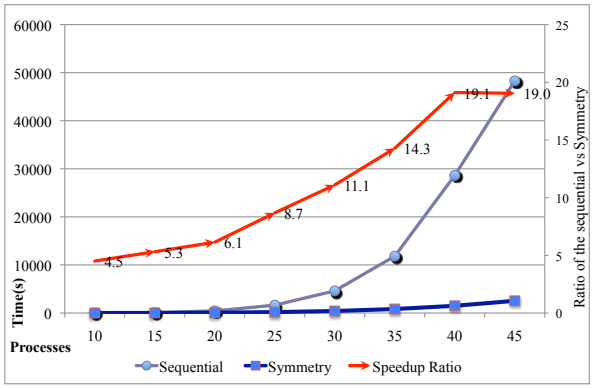
5.1 Symmetry

Figure 2 shows (a) the time spent in deadlock resolution, and (b) the total synthesis time for different numbers of processes in the Byzantine agreement problem. From this figure, we observe that the use of symmetry provides a remarkable improvement in the performance. More importantly, one can notice that the speedup ratio (gained using a symmetrical approach) grows with the increase in the number of processes. In particular, as shown in Figure 2 (b), the speedup ratio in the case of 10 non-general processes is 4.5 . However, in the case of 45 non-general processes the speedup ratio is 19. This behavior is both expected and highly valuable. Since symmetry uses transitions of one process to identify transitions of another process, it is expected that as the number of symmetric processes increases, so would the effectiveness of symmetry. Moreover, since the speedup is proportional to the number of (symmetric) processes, we argue that symmetry would be highly valuable in handling the state space explosion with an increased number of processes.

In Figure 3, we present the results of our experiments on the token ring problem. We observe that symmetry substantially reduces the time for deadlock resolution. In fact, symmetry was able to keep this time almost a constant, i.e., independent of the problem size. One can notice a spike in the required synthesis time of the sequential algorithm for token ring after we hit the threshold of 90 processes. This behavior was also observed in [4] and is caused by the fact that, at this state space, we are utilizing all the available memory, causing performance to degrade due to page faults.

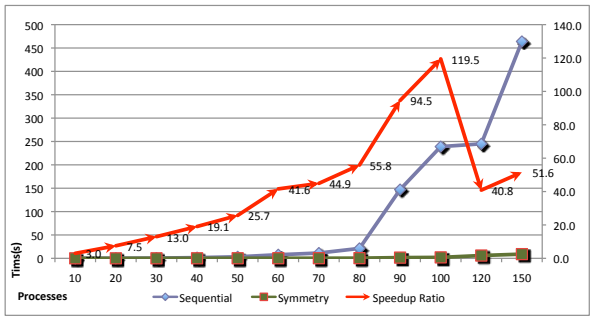


(a) Deadlock Resolution Time

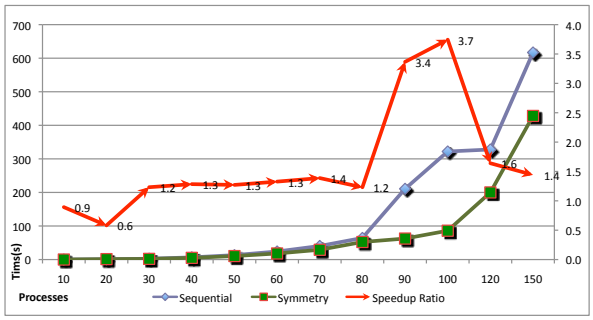


(b) Total Synthesis Time

Figure 2: The time required to (a) resolve deadlock states and (b) to synthesis tolerant program for several numbers of BA non-general processes in sequential and symmetrical algorithms. The BA has a state space $\approx 4 * 10^{1.08x}$ and reachable state space $\geq 2 * 10^{0.78x}$ where x is the number of process.



(a) Deadlock Resolution Time



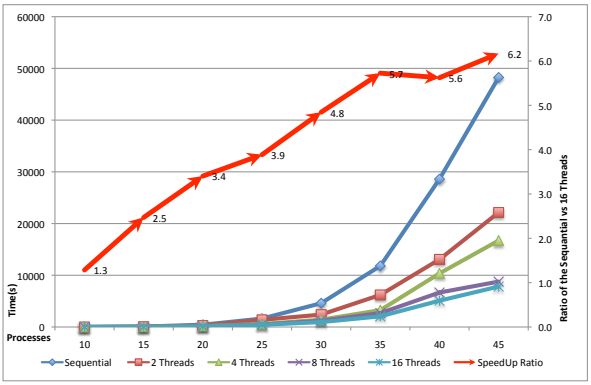
(b) Total Synthesis Time

Figure 3: The time required to (a) resolve deadlock states and (b) to synthesis tolerant program for several numbers of token ring processes in sequential and symmetrical algorithms. Token ring has a state space $\approx 4 * 10^{0.48x}$ and reachable state space $\geq 2 * 10^{0.3x}$ where x is the number of process.

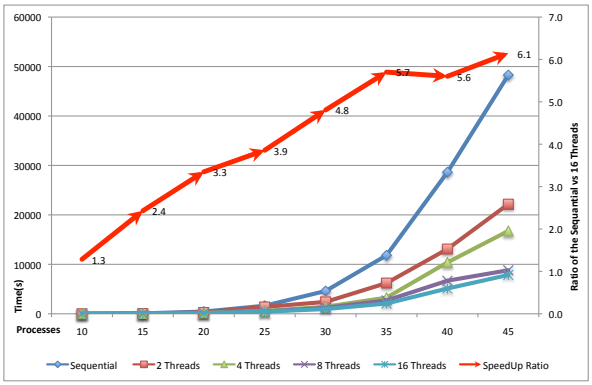
5.2 Parallelism

In Figure 4, we show the results of using the sequential approach versus the parallel approach (with multiple threads) to perform the synthesis. All the tests have shown that we gain a significant speedup. For example, in the case of 45 non-general processes and 8 threads we gain a speedup of 6.1 . We can clearly see that the parallel 16-thread version is faster than the the corresponding 8-thread version. This was surprising given that there are only 8 processors available. However, upon closer observation, we find that the group computation that is parallelized using threads is fine-grained. Hence, when the master thread uses multiple slave threads for performing the Group computation, the slave threads complete quickly and therefore cannot utilize the available resources to the full extent. Hence, creating more threads (than available processors) does improve the performance further.

In Figure 5, we present the results of our experiments in parallelizing the deadlock resolution of token ring problem. As described earlier in this section, after the number of processes exceeds a

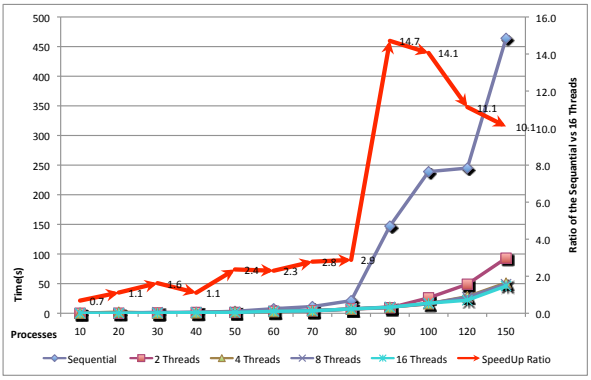


(a) Deadlock Resolution Time

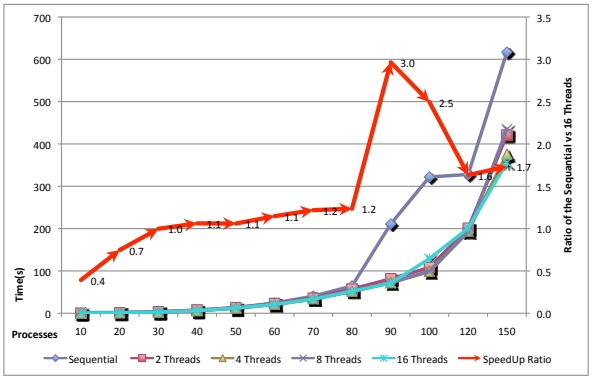


(b) Total Synthesis Time

Figure 4: The time required to (a) resolve deadlock states and (b) to synthesis tolerant program for several numbers of non-general processes of BA in sequential and parallel algorithms. The BA has a state space $\approx 4 * 10^{1.08x}$ and reachable state space $\geq 2 * 10^{0.78x}$ where x is the number of process.



(a) Deadlock Resolution Time

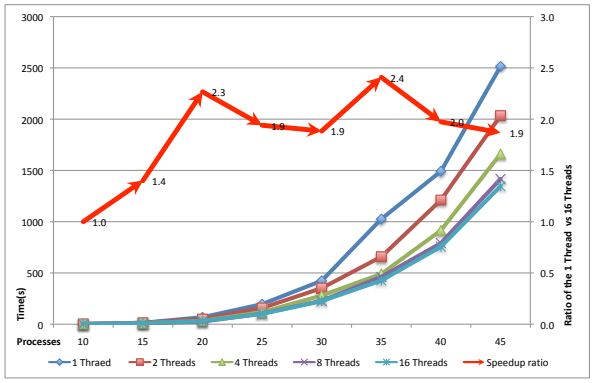


(b) Total Synthesis Time

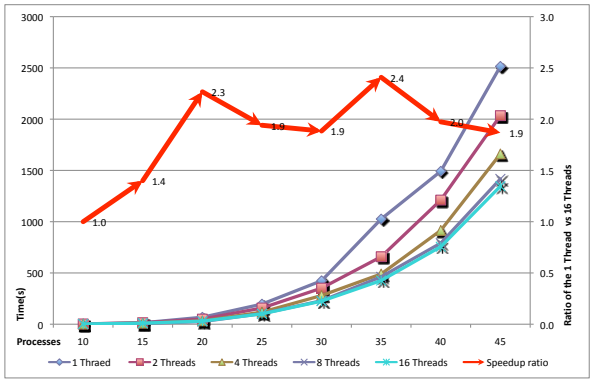
Figure 5: The time required to (a) resolve deadlock states and (b) to synthesis tolerant program for several numbers of token ring processes in Sequential and parallel algorithms. Token ring has a state space $\approx 4 * 10^{0.48x}$ and reachable state space $\geq 2 * 10^{0.3x}$ where x is the number of process.

threshold, the execution time increases substantially. This phenomenon also occurs in the case of parallelized implementation, although it appears for larger programs. However, this effect is not as strong. Note that the spike in speedup at 80 processes is caused by the page fault behavior where the performance of the sequential algorithm is affected although the performance of the parallel algorithm is still not affected.

Based on these results, we observe that symmetry outperforms parallelism in both of these examples. Recall that in parallelism we only divide the tasks assigned to one core among available cores. However, in the case of symmetry, we remove some computations from being performed where we generate transitions of several processes without performing an analysis of corresponding deadlock states.



(a) Deadlock Resolution Time



(b) Total Synthesis Time

Figure 6: The time required to (a) resolve deadlock states and (b) to synthesis tolerant program for several numbers of non-general processes of BA in Symmetrical and parallel Symmetrical algorithms. The BA has a state space $\approx 4 * 10^{1.08x}$ and reachable state space $\geq 2 * 10^{0.78x}$ where x is the number of process.

5.3 Symmetry and Parallelism

In this subsection, we present our experimental results of using parallelism in computing the symmetry. The results of parallelizing the symmetry computation with various implementations in the automated symbolic synthesis are presented in Figure 6. We have achieved the shortest synthesis time when we used parallelism to compute the symmetry. For example, in the case of the Byzantine agreement with 45 non-general processes using 16 threads, we achieve a speedup ratio of 1.8 times that of the symmetry alone. Since in case of the token ring, symmetry alone reduces the time of computing recovery transitions to a negligible amount, the results for this case are omitted.

5.4 Memory Usage

Both of our approaches, symmetry and parallelism, require the use of more memory. For instance, the synthesis of the *BA* with 2 threads requires almost twice the amount of memory need by the sequential algorithm for the same number of non-general processes. However, unlike model checking, in synthesis, since we always run out of time before we run out of memory, we argue that the extra usage of memory is acceptable given the remarkable reductions we achieve in total synthesis time.

6 Related Work

Automated program synthesis and revision has been studied from various perspectives. Inspired by the seminal work by Emerson and Clarke [8], Arora, Attie, and Emerson [1] propose an algorithm for synthesizing fault-tolerant programs from CTL specifications. Their method, however, does not address the issue of the addition of fault-tolerance to existing programs. Kulkarni and Arora [14] introduce enumerative synthesis algorithms for automated addition of fault-tolerance to centralized and distributed programs. In particular, they show that the problem of adding fault-tolerance to distributed programs is NP-complete. In order to remedy the NP-hardness of the synthesis of fault-tolerant distributed programs and overcome the state explosion problem, we proposed a set of symbolic heuristics [4], which allowed us to synthesize programs with a state space size of 10^{30} and beyond.

Ebnenasir [7] presents a divide-and-conquer method for synthesizing *failsafe* fault-tolerant distributed programs. A failsafe program is one that does not need to satisfy its liveness specification

in the presence of faults. Thus, a respective synthesis algorithm does not need to resolve deadlock states outside the invariant predicate. Moreover, Ebnenasir's synthesis method resolves deadlock states inside the invariant predicate in a sequential manner.

Parallelization of symbolic reachability analysis has been studied in the model checking community from different perspectives. In [9–11], the authors propose solutions and analyze different approaches to parallelization of the *saturation*-based generation of state space in model checking. In particular, in [10], the authors show that in order to gain speedups in saturation-based parallel symbolic verification, one has to pay a penalty for memory usage of up to 10 times, that of the sequential algorithm. Other efforts range from simple approaches that essentially implement BDDs as two-tiered hash tables [17, 19], to sophisticated approaches relying on *slicing* BDDs [13] and techniques for *workstealing* [12]. However, the resulting implementations show only limited speedups.

In [3], we have identified an approach to parallelize deadlock resolution where the set of available deadlock states are partitioned and multiple threads are used to resolve them. However, in this approach, the tasks performed by different threads may conflict with each other, e.g., one thread may choose to remove a transition while another may require that it be included. Hence, the results from the parallel threads need to be merged to remove such inconsistencies. Thus, the parallelism is coarse-grained. By contrast, in this work, the parallelism is fine-grained when a given task of *Group* computation is achieved by cooperation of multiple threads. These two approaches are orthogonal and there is a potential to combine them.

7 Conclusion and Future Work

In this paper, we focused on improving the synthesis of fault-tolerant programs from their fault-intolerant version. We focused on two aspects from automated program verification: (1) use of symmetry and (2) use of parallelism. We showed that symmetry provides a substantial benefit in reducing the time involved in synthesis. Moreover, the speedup increases as the number of symmetric processes increases.

We also showed that the use of multiple threads to parallelize the synthesis algorithm reduces the time substantially. Since the configuration used to evaluate performance was on an 8-core (4 dual-cores) machine, we considered the case where up to 16 threads are used. We find that as the number of threads increases, the synthesis time decreases. In fact, because the parallelism is fine-grained, using more threads than available cores has the potential to improve the performance slightly. This demonstrates that we have not yet reached the bottleneck involved in parallelization; one future work in this area is to evaluate these algorithms on a machine with larger number of processors to identify these bottlenecks.

We believe that the results from this work are especially important to the distributed system community where the difficulty of designing distributed programs with all its non-determinism and race conditions is well understood. Hence, the ability to automate (to the extent possible) fault-tolerant distributed programs is very important. This work demonstrates that using techniques from the automated verification community and parallel processing community has the potential to reduce the cost of automating the design of distributed programs. Also the approaches used in this paper, the use of symmetry between different processes and the parallelism of group computation that is caused by partial knowledge of the process, are designed for the main characteristics of distributed programs. One future work in this context is to combine other advances from program verification. We expect that by combining these advances along with characteristics of distributed systems, e.g., hierarchical behavior, types of expected faults, etc., it would become feasible for the automated revision of practical distributed programs to add new properties.

Based on the results in this paper, there is potential for further reduction in synthesis time if the level of parallelism is increased (e.g., if there are more processors). Although the level of parallelism is fine-grained, we showed that the overhead of parallel computation is small. Hence, another future

work is to evaluate the limits of parallel computation in improving performance of the synthesis algorithm and include this in the tools (e.g., SYCRAFT [5]) for synthesizing fault-tolerance.

Another future work is to combine the parallelism in this work with that in [3]. In particular, as discussed in Section 6, the parallelism in [3] is coarse-grained. However, it can permit threads to perform inconsistent behavior that needs to be resolved later. Thus, it provides a tradeoff between overhead of synchrony among threads and potential error resolutions. Thus, one of the future work is to combine the use of symmetry and fine-grain parallelism in this work with coarse-grained parallelism from [3].

References

[1] A. Arora, P. C. Attie, and E. A. Emerson. Synthesis of fault-tolerant concurrent programs. In *Principles of Distributed Computing (PODC)*, pages 173–182, 1998.

[2] A. Arora and S. S. Kulkarni. Component based design of multitolerant systems. *IEEE Transactions on Software Engineering*, 24(1):63–78, 1998.

[3] B. Bonakdarpour, F. Abujarad, and S. S. Kulkarni. Parallelizing deadlock resolution in symbolic synthesis of distributed programs. Technical Report MSU-CSE-08-25, Department of Computer Science and Engineering, Michigan State University, 2008.

[4] B. Bonakdarpour and S. S. Kulkarni. Exploiting symbolic techniques in automated synthesis of distributed programs with large state space. In *IEEE International Conference on Distributed Computing Systems (ICDCS)*, pages 3–10, 2007.

[5] B. Bonakdarpour and S. S. Kulkarni. Sycraft: A tool for automated synthesis of fault-tolerant distributed programs. *International Conference on Concurrency Theory (CONCUR)*, 2008.

[6] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[7] A. Ebnenasir. DiConic addition of failsafe fault-tolerance. In *Automated Software Engineering (ASE)*, pages 44–53, 2007.

[8] E. A. Emerson and E. M. Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2(3):241–266, 1982.

[9] J. Ezekiel and G. Lüttgen. Measuring and evaluating parallel state-space exploration algorithms. In *International Workshop on Parallel and Distributed Methods in Verification (PDMC)*, 2007.

[10] J. Ezekiel, G. Lüttgen, and G. Ciardo. Parallelising symbolic state-space generators. In *Computer Aided Verification (CAV)*, pages 268–280, 2007.

[11] J. Ezekiel, G. Lüttgen, and R. Siminiceanu. Can Saturation be parallelised? on the parallelisation of a symbolic state-space generator. In *International Workshop on Parallel and Distributed Methods of Verification (PDMC)*, pages 331–346, 2006.

[12] O. Grumberg, T. Heyman, N. Ifergan, and A. Schuster. Achieving speedups in distributed symbolic reachability analysis through asynchronous computation. In *Correct Hardware Design and Verification Methods (CHARME)*, pages 129–145, 2005.

[13] O. Grumberg, T. Heyman, and A. Schuster. A work-efficient distributed algorithm for reachability analysis. *Formal Methods in System Design (FMSD)*, 29(2):157–175, 2006.

[14] S. S. Kulkarni and A. Arora. Automating the addition of fault-tolerance. In *Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT)*, pages 82–93, 2000.

[15] S. S. Kulkarni and A. Ebnenasir. The effect of the specification model on the complexity of adding masking fault-tolerance. *IEEE Transactions on Dependable and Secure Computing*, 2(4):348–355, 2005.

[16] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

[17] K. Milvang-Jensen and A. J. Hu. BDDNOW: A parallel BDD package. In *Formal Methods in Computer Aided Design (FMCAD)*, pages 501–507, 1998.

[18] F. Somenzi. CUDD: Colorado University Decision Diagram Package. `http://vlsi.colorado.edu/~fabio/CUDD/cuddIntro.html`.

[19] T. Stornetta and F. Brewer. Implementation of an efficient parallel BDD package. In *Design automation (DAC)*, pages 641–644, 1996.